Algorithms for Collision Detection Between a Point and a Moving Polygon, with Applications to Aircraft Weather Avoidance

Anthony Narkawicz * George Hagen *

This paper proposes mathematical definitions of functions that can be used to detect future collisions between a point and a moving polygon. The intended application is weather avoidance, where the given point represents an aircraft and bounding polygons are chosen to model regions with bad weather. Other applications could possibly include avoiding other moving obstacles. The motivation for the functions presented here is *safety*, and therefore they have been proved to be mathematically correct. The functions are being developed for inclusion in NASA's Stratway software tool, which allows low-fidelity air traffic management concepts to be easily prototyped and quickly tested.

The significant disruptions in scheduling and flight planning that can be caused by bad weather present a challenge for air traffic management systems, due to the uncertain behavior of weather systems.[2] In fact, weather is responsible for 70% of all delays. Convective weather can cause significant challenges in air traffic control, in part because tools used in normal conditions are no longer used in the presence of convective weather.[6] Using systems that aid pilots in avoiding weather creates some safety challenges due to the potential dangers of failing to avoid convective weather. Weather can be dangerous for an aircraft. FAA Advisory Circular 00-24C states that "Knowledge of thunderstorms and the associated hazards with thunderstorms is critical to the safety of flight... Weather recognizable as a thunderstorm should be considered hazardous, as penetration of any thunderstorm can lead to an aircraft accident and fatalities to those on board".[1]

Thus, any viable air traffic management system must reliably integrate weather awareness and avoidance. In fact, there are numerous bodies of work documenting this integration for a variety of systems.[2,6] These approaches commonly model regions of bad weather with polygons that over approximate the weather cells. This modeling choice is made because the mathematics involved in avoiding a polygon is usually simpler than avoiding other objects, such as those defined by a simple closed curves,[3] although modeling the avoidance regions with overlapping circles also produces relatively simple mathematics.

It is therefore desirable to have simple, easy-to-implement methods for determining whether an aircraft will encounter a weather polygon in the near future, along its current trajectory. If the aircraft is modeled as a point mass, and it is expected to follow a linear trajectory for the near future (e.g. a few minutes), then it suffices to use a function that detects whether a linearly moving point in space will intersect a polygon within a specified time (called a *lookahead time*). One way to define such a function is to project the position of the point ahead in one-second increments and test whether, at each of these points, the point is inside the polygon. In fact, well-known analytic algorithms do already exist for testing whether a point is inside a polygon, the most common being *ray casting* and *winding number* algorithms.[8] Collision detection by incrementing time and checking containment has some drawbacks, including the facts that it can be computationally inefficient and that by skipping ahead in time increments, small violations can be missed.

Rather than incrementally stepping forward in time and checking containment in a polygon, there are available analytic ways to detect collisions involving polygons.[4] Some of these methods detect collisions between two statically-shaped moving polygons, rather than a polygon and a moving point.

*NASA Langley Research Center, Hampton, VA 23681, USA

This paper presents analytic functions for detecting collisions between a linearly moving point (e.g. aircraft) and a (possibly moving) polygon in 3 dimensions. They work for a 3 dimensional polygon formed from a 2 dimensional polygon and maximum/minimum altitudes. Several of the detection functions presented have been proved to be mathematically correct, assuming there are no floating point errors, in the Prototype Verification System (PVS).[5] They are being developed for inclusion in NASA's Stratway software tool, which allows low-fidelity air traffic management concepts to be easily prototyped and quickly tested.

I. Mathematical Background

This paper assumes that accurate position information is available as horizontal and vertical components in a three dimensional (3D) space. Letters in **bold-face** denote two dimensional (2D) vectors. Vector operations such as addition, subtraction, scalar multiplication, dot product, i.e., $\mathbf{s} \cdot \mathbf{v} = \mathbf{s}_x\mathbf{v}_x + \mathbf{s}_y\mathbf{v}_y$, the square of a vector, i.e., $\mathbf{s}^2 = \mathbf{s}\cdot\mathbf{s}$, and the norm of a vector, i.e., $\|\mathbf{s}\| = \sqrt{\mathbf{s}_x^2 + \mathbf{s}_y^2}$, are defined in a 2D Euclidean geometry. Furthermore, the expression $\mathbf{v}^\perp$ denotes the 2D right perpendicular of $\mathbf{v}$, i.e., $\mathbf{v}^\perp = (\mathbf{v}_y, \mathbf{v}_x)$, and $\mathbf{0}$ denotes the 2D vector whose components are 0, i.e., $\mathbf{0} = (0, 0)$. Further, `sign` is the function such that for any real number x, $\texttt{sign}(x) = 1$ when $x \geq 0$ and $\texttt{sign}(x) = -1$ when $x < 0$.

In later sections, quadratics (polynomials of degree 2) arise in one variable and two variables. In this section, two functions are presented that determine whether a given quadratic ever attains a value below a given threshold value D when its input values are restricted to the unit interval. For a univariate quadratic

$$\texttt{quad}(a, b, c)(t) = at^2 + bt + c,$$

the method defined below determines if it ever takes a value less than D for t in the interval $[0, 1]$.

`quad_min_le_D`(a, b, c, D)**:**

1. If $\texttt{quad}(a, b, c)(0) < D$ or $\texttt{quad}(a, b, c)(1) < D$, then return *true*
2. If $a > 0$ and $b \leq 0$ and $-b \leq 2a$ and $b^2 - 4a(c - D) > 0$, then return *true*
3. Otherwise, return *false*

The function call `quad_min_le_D`(a, b, c, D) returns *true* if and only if there exists some $t \in [0, 1]$ such that $\texttt{quad}(a, b, c)(t) < D$.

This paper also considers a quadratic in two variables:

$$\texttt{quad2D}(a, b, c, d, e, f)(r, t) = ar^2 + bt^2 + crt + dr + et + f$$

The following method determines if this function ever takes a value less than D for r and t restricted to the unit inverval $[0, 1]$.

`quad_min_box_le_D`(a, b, c, d, e, f, D)**:**

1. If $\texttt{quad\_min\_le\_D}(a, c + d, \texttt{quad}(b, e, f)(1), D) = \textit{true}$, then return *true*
2. If $\texttt{quad\_min\_le\_D}(a, d, f, D) = \textit{true}$, then return *true*
3. If $\texttt{quad\_min\_le\_D}(b, c + e, \texttt{quad}(a, d, f)(1), D) = \textit{true}$, then return *true*
4. If $\texttt{quad\_min\_le\_D}(b, e, f, D) = \textit{true}$, then return *true*
5. Set $\texttt{disc} = c^2 - 4ab$, $\texttt{mx} = 2bd - ce$, $\texttt{my} = 2ae - cd$
6. Return *true* if each of the following conditions is met
   - $0 \leq \texttt{mx} \cdot \texttt{disc} \leq \texttt{disc}^2$
   - $0 \leq \texttt{my} \cdot \texttt{disc} \leq \texttt{disc}^2$

- $\texttt{quad2D}(a, b, c, d \cdot \texttt{disc}, e \cdot \texttt{disc}, f \cdot \texttt{disc}^2)(\texttt{mx}, \texttt{my}) < D \cdot \texttt{disc}^2$

7. Otherwise, return *false*

It can be proved mathematically that the function call $\texttt{quad\_min\_box\_le\_D}(a, b, c, d, e, f, D)$ returns *true* if and only if there exist r and t in the unit inverval $[0, 1]$ such that $\texttt{quad2D}(a, b, c, d, e, f)(r, t) < D$.

II. Detection for 2D Polygons

This section presents detection functions for collisions between a point and a (possibly) moving 2D polygon. These functions are used later in Section III, where detection functions are presented for 3D polygons. Each 2D polygon is specified simply as a sequence $\boldsymbol{P} = \{\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n\}$ of points in the horizontal plane $\mathbf{R}^2$, each having an x and a y coordinate. If it is moving, then there is a sequence $\boldsymbol{V} = \{\boldsymbol{v}_0, \ldots, \boldsymbol{v}_n\}$ of velocities in $\mathbf{R}^2$, one velocity for each vertex of $\boldsymbol{P}$. At any time t in the future, the vertices of the 2D polygon are given by the sequence

$$\boldsymbol{P} + t \cdot \boldsymbol{V} \;=\; \{\boldsymbol{p}_0 + t \cdot \boldsymbol{v}_0,\; \ldots, \boldsymbol{p}_n + t \cdot \boldsymbol{v}_n\}.$$

This notation is also used in Section III when referring to the 2D component of a 3D polygon.

Loosely speaking, detecting collisions during a given lookahead time interval $[0, T]$, between a moving horizontal position $\mathbf{s}$ with velocity $\mathbf{v}$ and a moving 2D polygon $\boldsymbol{P} = \{\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n\}$ with velocity sequence $\boldsymbol{V} = \{\boldsymbol{v}_0, \ldots, \boldsymbol{v}_n\}$, is accomplished in two steps:

1. Detect whether $\mathbf{s}$ is inside $\boldsymbol{P}$. If so, return *true*.
2. If not, detect whether the moving point collides with any of the moving edges of the polygon during $[0, T]$. That is, detect whether $\mathbf{s} + t \cdot \mathbf{v}$ is ever on the line segment between $\boldsymbol{p}_i + t \cdot \boldsymbol{v}_i$ and $\boldsymbol{p}_{i+1} + t \cdot \boldsymbol{v}_{i+1}$ for any i such that $0 \le i \le n$. If so, return *true*.
3. Otherwise, return *false*.

However, the method above is *not quite* the method used in this paper for the following reasons. As it turns out, in the presence of position errors or possible floating point errors in the computations, computing whether either (1) or (2) occurs *exactly* can be somewhat challenging.[7] There are two key challenges:

- If the point $\mathbf{s}$ is very close to an edge of $\boldsymbol{P}$, small errors can cause an incorrect computation of containment.
- Detecting whether a point is on a line segment often involves *equality* symbols ($=$), which in the presence of position or floating point errors, will almost always return false, making this detection unreliable.

The detection functions in this paper use a pragmatic approach to deal with these challenges that can arise due to data errors. The first is handled by introducing a small buffer distance `BUFF`, described below, around the edges of the polygon. The second is handled by introducing a small fraction F, representing the fraction of the length of an edge to extend a boundary to search for collisions with the moving point.

II.A. Ray Casting

The containment method for 2D polygons assumes that any input polygon is arranged in *counterclockwise* order. The function `definitely_outside`, defined later, uses a standard and well-known method of determining whether a point lies outside a polygon, namely *ray casting*.[8] A ray is cast from the point outward to infinity (in this case the direction of the positive y-axis), and in most cases, if it crosses an even number of edges of the polygon, it is outside; otherwise, it is inside. This is a standard approach for testing containment in a polygon, which is shown in Figure 1.

In the containment method for 2D polygons, a *buffer* named `BUFF` is introduced so that any point within this distance of an edge will automatically be considered inside. The distance `BUFF` is also used to perturb

the original polygon $\boldsymbol{P}$ when it is configured because standard *ray casting*[8] along the direction of the y axis can sometimes cause the ray to pass very close to some vertices. The perturbation of these vertices by BUFF stops this from happening, and the ray casting function then works as expected.

One effect of introducing small buffers around the polygon edges like this is that the functions will declare some points, which are near an edge of the polygon, as being neither definitely inside nor definitely outside. This may make the detection algorithms return *true* in some cases where there is no future collision between the trajectory and the polygon but where the trajectory will come within the small buffer distance of an edge. This results in a *false detection*. The alternative might allow data errors to cause some *missed detections*. Given that one purpose of these functions is safety, false detections are more desirable than missed detections.

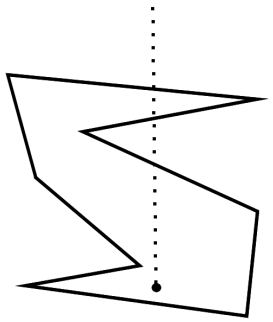

Figure 1. Ray Casting

II.B. Perturbing P to $P^*_{\texttt{BUFF}}$ to Make Ray Casting More Reliable

A problem can occur in a ray casting function when the infinite ray that is cast either crosses a vertex or comes close to a vertex. In this case, floating point errors can possibly cause the function to count the wrong number of crossing points, reversing the correct inside/outside determintation. This is mitigated in the functions presented here by first *perturbing* the polygon $\boldsymbol{P}$ by slightly moving any vertices which have x coordinates within BUFF of $\mathbf{s}$. The perturbation process is a standard method to address this problem. It returns a new polygon $\boldsymbol{P}^*_{\texttt{BUFF}}$, and it is described as follows.

$\boldsymbol{P} \longrightarrow \boldsymbol{P}^*_{\texttt{BUFF}}$:

1. If $\boldsymbol{p}_{iy} \geq \mathbf{s}_y - \texttt{BUFF}$ and $|\boldsymbol{p}_{ix} - \mathbf{s}_x| \leq \texttt{BUFF}$, then the i-th vertex of $\boldsymbol{P}^*_{\texttt{BUFF}}$ is $p(i) - (2\texttt{BUFF}, 0)$
2. Otherwise, the i-th vertex of $\boldsymbol{P}^*_{\texttt{BUFF}}$ is $p(i)$.

II.C. Checking Proximity to an Edge with `near_edge`

The function that determines if a point is definitely outside the polygon excludes any point that is within the buffer distance BUFF of any edge of either $\boldsymbol{P}$ or the perturbed polygon $\boldsymbol{P}^*_{\texttt{BUFF}}$. To check if $\mathbf{s}$ is within BUFF of some point on the segment between the two points $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$, note that every point on this line segment is equal to $\boldsymbol{p}_i + r \cdot (\boldsymbol{p}_j - \boldsymbol{p}_i)$ for some real number r in the interval $[0, 1]$. Thus, $\mathbf{s}$ is within distance BUFF of some point on the segment if and only if there exists $r \in [0, 1]$ such that the following (in)-equalities hold.

$$\|(\boldsymbol{p}_i + r \cdot (\boldsymbol{p}_j - \boldsymbol{p}_i)) - \mathbf{s}\|^2 = (\boldsymbol{p}_j - \boldsymbol{p}_i)^2 \cdot r^2 + (2(\boldsymbol{p}_i - \mathbf{s}) \cdot (\boldsymbol{p}_j - \boldsymbol{p}_i)) \cdot r + (\boldsymbol{p}_i - \mathbf{s})^2 < \texttt{BUFF}^2$$

The second term (following the equality sign =) is a *quadratic* in r, and it is simple to determine analytically, using the function `quad_min_le_D` from Section I, whether this quadratic is ever less than $\texttt{BUFF}^2$ on the interval $[0, 1]$. This motivates the definition of the function `near_edge`.

`near_edge`$(\boldsymbol{p}_i, \boldsymbol{p}_j, \mathbf{s}, \texttt{BUFF})$:

1. If $\mathbf{s}$ is within `BUFF` of $\boldsymbol{p}_i$ or $\boldsymbol{p}_j$ ($\|\mathbf{s}-\boldsymbol{p}_i\|^2 < \texttt{BUFF}^2$ or $\|\mathbf{s}-\boldsymbol{p}_j\|^2 < \texttt{BUFF}^2$), then return *true*.
2. If $\boldsymbol{p}_i \neq \boldsymbol{p}_j$, then define a polynomial ar^2+br+c by setting $a=(\boldsymbol{p}_j-\boldsymbol{p}_i)^2$, $b=(2(\boldsymbol{p}_i-\mathbf{s})\cdot(\boldsymbol{p}_j-\boldsymbol{p}_i))$, and $c=(\boldsymbol{p}_i-\mathbf{s})^2$. If `quad_min_le_D`$(a,b,c,\texttt{BUFF}^2)$ returns *true*, then return *true*.
3. Otherwise, return *false*.

It is relatively easy to prove that the function `near_edge` returns true precisely when $\mathbf{s}$ has distance less than `BUFF` from at least one point on the segment between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$.

II.D. Counting the Number of Ray Crosses Using `num_cross`

The function that counts the number of edges of $\boldsymbol{P}$ crossing the infinite ray computes, for each edge between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$, whether the segment between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ crosses the infinite ray $\{(\mathbf{s}_x, \mathbf{s}_y+z) \in \mathbf{R}^2 | z \geq 0\}$ from $\mathbf{s}$ in the direction of the positive y-axis. Any point $\mathbf{q}$ on the segment between these two vertices satisfies $(\mathbf{q}-\boldsymbol{p}_i)\cdot(\boldsymbol{p}_j-\boldsymbol{p}_i)^\perp = 0$. Thus, if the point $(\mathbf{s}_x, \mathbf{s}_y+z)$ is on this segment and $z \geq 0$ then

$$(\mathbf{s}-\boldsymbol{p}_i)\cdot(\boldsymbol{p}_j-\boldsymbol{p}_i)^\perp - z\,(\boldsymbol{p}_{jx}-\boldsymbol{p}_{ix}) = 0.$$

If $\boldsymbol{p}_{jx} \neq \boldsymbol{p}_{ix}$, then this means that

$$z = \frac{(\mathbf{s}-\boldsymbol{p}_i)\cdot(\boldsymbol{p}_j-\boldsymbol{p}_i)^\perp}{\boldsymbol{p}_{jx}-\boldsymbol{p}_{ix}}.$$

Plugging this into the equation $z \geq 0$ and multiplying by $(\boldsymbol{p}_{jx}-\boldsymbol{p}_{ix})^2$ gives

$$(\boldsymbol{p}_{jx}-\boldsymbol{p}_{ix})\cdot((\mathbf{s}-\boldsymbol{p}_i)\cdot(\boldsymbol{p}_j-\boldsymbol{p}_i)^\perp) \geq 0.$$

It turns out that this inequality is sufficient to test whether the infinite ray crosses the segment between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$, but only when $\boldsymbol{p}_{ix}$ and $\boldsymbol{p}_{jx}$ are not on the *same side* of $\mathbf{s}_x$.

The function `num_cross`$(\boldsymbol{P}, \mathbf{s})$ is described below. It is an implicit assumption in this function that the polygon is already *perturbed* in the sense of Section II.B. That is, it is assumed that there are no vertices of $\boldsymbol{P}$ on the infinite ray $\{(\mathbf{s}_x, \mathbf{s}_y+z) \in \mathbf{R}^2 | z \geq 0\}$. In other words, $\boldsymbol{P}$ has no point $\boldsymbol{p}_i$ such that $\boldsymbol{p}_{ix} = \mathbf{s}_x$ and $\boldsymbol{p}_{iy} \geq \mathbf{s}_y$. The function `num_cross`$(\boldsymbol{P}, \mathbf{s})$ counts each edge that crosses the infinite ray from $\mathbf{s}$. For each edge (in counterclockwise order) from $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$ (so either $j=i+1$ or $i=n$ and $j=0$), the following method counts the edges crossing the infinite ray from $\mathbf{s}$.

`num_cross`$(\boldsymbol{P}, \mathbf{s})$**:**

- Count the number of edges $\boldsymbol{p}_i/\boldsymbol{p}_j$ of $\boldsymbol{P}$ crossing the infinite ray from $\mathbf{s}$ using the following criteria.
  - If $\boldsymbol{p}_{ix} \geq \mathbf{s}_x$ and $\boldsymbol{p}_{jx} \geq \mathbf{s}_x$, then the edge does not cross the ray
  - If $\boldsymbol{p}_{ix} < \mathbf{s}_x$ and $\boldsymbol{p}_{jx} < \mathbf{s}_x$, then the edge does not cross the ray
  - If $\boldsymbol{p}_{ix} = \boldsymbol{p}_{jx}$, then the edge does not cross the ray
  - If $(\boldsymbol{p}_{jx}-\boldsymbol{p}_{ix})\cdot((\mathbf{s}-\boldsymbol{p}_i)\cdot(\boldsymbol{p}_j-\boldsymbol{p}_i)^\perp) \geq 0$, then the edge crosses the ray
  - Otherwise, the edge does not cross the ray

It is most important to note that this method will not necessarily work if the polygon is not already perturbed away from the infinite array as in Section II.B.

II.E. Determine if a Point is Definitely Outside Using `definitely_outside`

The function that determines whether the point $\mathbf{s}$ is definitely outside $\boldsymbol{P}$ is defined as follows. It uses the standard and well-known ray casting method[8] described in Sections II.A and II.D.

`definitely_outside`$(\boldsymbol{P}, \mathbf{s}, \texttt{BUFF})$**:**

1. Compute the perturbed polygon $\boldsymbol{P}^*_{\texttt{BUFF}}$.
2. Use the function `near_edge` to determine if $\mathbf{s}$ is within `BUFF` of any edge of $\boldsymbol{P}$. If so, return *false*.
3. Similarly use `near_edge` to determine if $\mathbf{s}$ is within `BUFF` of any edge of $\boldsymbol{P}^*_{\texttt{BUFF}}$. If so, return *false*.
4. If `even?`(`num_cross`$(\boldsymbol{P}^*_{\texttt{BUFF}}, \mathbf{s})$), then return *true*
5. Otherwise, return *false*

II.F. Two Types of 2D Detection Methods

Recall that detecting collisions with a moving polygon is decomposed into two steps: (1) Determining whether the point is inside the polygon at time 0, and (2) Detecting whether the moving point collides with any of the moving edges during the lookahead time interval. The first of these steps is described above. The next two sections focus on the other step, namely detecting collisions with moving segments, where each vertex of the segments has a velocity. There are two cases here:

- The vertices of the polygons all have the *same* velocity vector. In this case the polygon is *statically shaped*.
- The vertices of the polygons may have *different* velocity vectors. In this case the polygon is *warping*.

This section presents two methods, one for each of these cases, that solve the detection problem. The function `detect_same_2D` detects collisions between the moving point and a moving polygon with the same velocity for each vertex, and the function `detect_diff_2D` detects collisions where the vertices may have different velocity vectors. In each case, rather than detecting if there is an actual collision, the functions detect whether the moving point comes *close* to the moving polygon.

II.G. Determining if Two Segments are Close Using `segments_close`

Detection for statically shaped polygons will use a function that determines whether two line segments are within distance `BUFF` of each other at their closest points. This is handled by the function `segments_close`, defined below, which determines whether the line segment between $\mathbf{c}$ and $\mathbf{d}$ comes within distance `BUFF` of the line segment between $\mathbf{e}$ and $\mathbf{f}$.

`segments_close`$(\mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}, \texttt{BUFF})$**:**

1. Use `near_edge` to determine if $\mathbf{c}$ or $\mathbf{d}$ is within `BUFF` of the edge between $\mathbf{e}$ and $\mathbf{f}$. If so, return *true*.
2. Similarly use `near_edge` to determine if $\mathbf{e}$ or $\mathbf{f}$ is within `BUFF` of the edge between $\mathbf{c}$ and $\mathbf{d}$. If so, likewise return *true*.
3. Set $\mathbf{q} = \mathbf{c} - \mathbf{e}$, $\mathbf{u} = \mathbf{d} - \mathbf{c}$, and $\mathbf{w} = \mathbf{f} - \mathbf{e}$.
4. If `quad_min_box`$(\mathbf{u}\cdot\mathbf{u}, \mathbf{w}\cdot\mathbf{w}, -2\cdot\mathbf{u}\cdot\mathbf{w}, 2\cdot\mathbf{q}\cdot\mathbf{u}, -2\cdot\mathbf{q}\cdot\mathbf{w}, \mathbf{q}\cdot\mathbf{q}, \texttt{BUFF}\cdot\texttt{BUFF})$ returns *true*, then return *true*.
5. Otherwise, return *false*.

The function `quad_min_box_le_D` is defined earlier in Section I.

II.H. Detection for Statically Shaped Polygons Using `detect_same_2D`

Detection for statically shaped polygons, i.e., those with the same velocity for every vertex, is much simpler than for warping edges. As above, let $\mathbf{s}$ be the position of the point at time 0, and assume it has velocity $\mathbf{v}$. Consider an edge (segment) of the polygon $\boldsymbol{P}$ between the vertices $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$. Note that this means that either $i = j + 1$, $j = i + 1$, or that one of j and i is 0 and the other is n. Suppose that $\boldsymbol{v}_i = \boldsymbol{v}_j$ (the vertices have the same velocity). Then the segment between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ does not change length or direction as time progresses. If there is a time $t \in [0, T]$ such that $\mathbf{s} + t \cdot \mathbf{v}$ is within distance `BUFF` of some point q on the line segment between $\boldsymbol{p}_i + t \cdot \boldsymbol{v}_i$ and $\boldsymbol{p}_j + t \cdot \boldsymbol{v}_j$, then since $\boldsymbol{v}_i = \boldsymbol{v}_j$, there is some real number r in $[0, 1]$ such that $q = (\boldsymbol{p}_i + t \cdot \boldsymbol{v}_i) + r \cdot ((\boldsymbol{p}_j + t \cdot \boldsymbol{v}_i) - (\boldsymbol{p}_i + t \cdot \boldsymbol{v}_i)) = (\boldsymbol{p}_i + r \cdot (\boldsymbol{p}_j - pi)) + t \cdot \boldsymbol{v}_i$. Thus, since $\mathbf{s} + t \cdot \mathbf{v}$ is within distance `BUFF` of q,

$$\texttt{BUFF}^2 \; > \; (\mathbf{s} - q)^2 \; = \; ((\mathbf{s} + t \cdot (\mathbf{v} - \boldsymbol{v}_i)) - (\boldsymbol{p}_i + r \cdot (\boldsymbol{p}_j - pi)))^2.$$

This means that the point $\mathbf{s} + t \cdot (\mathbf{v} - \mathbf{v}_i)$ is within distance `BUFF` of the static (non-moving) line segment between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$. Stated another way, the line segment between $\mathbf{s}$ and $\mathbf{s} + T \cdot (\mathbf{v} - \boldsymbol{v}_i)$ is within distance `BUFF` of the line segment between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$. Thus, detection for statically shaped edges is easily reduced to the simpler problem of determining whether two stationary segments are within a given distance of each other. This is easily computed using the function `segments_close` as described in Section II.G. This therefore enables the definition of `detect_same_2D` for detecting collisions (or proximity) between a moving point and a moving polygon with equal velocities at its endpoints.

`detect_same_2D`$(T, \mathbf{s}, \mathbf{v}, \boldsymbol{P}, \boldsymbol{V}, \texttt{BUFF})$**:**

1. Using the function `definitely_outside`, defined above, check whether the point $\mathbf{s}$ is definitely outside $\boldsymbol{P}$. If not, then return *true*.
2. Using `definitely_outside`, check whether $\mathbf{s} + T \cdot \mathbf{v}$ is definitely outside $\boldsymbol{P} + T \cdot \boldsymbol{V}$. If not, then return *true*.
3. For each edge from $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, with velocity $\boldsymbol{v}_i$, calculate `segments_close`$(\mathbf{s}, \mathbf{s} + T \cdot (\mathbf{v} - \boldsymbol{v}_i), \boldsymbol{p}_i, \boldsymbol{p}_j, \texttt{BUFF})$. If this returns *true* for any edge, then return *true*
4. Otherwise, return *false*

II.I. Detection for Warping Polygons Using `detect_diff_2D`

Detection for warping polygons, i.e., those with possibly different velocities for each vertex, is slightly more complicated than for statically shaped polygons. However, the mathematics involved is still simple, depending only on basic algebra. One significant difference is that a linearly moving point can meet the same warping edge twice, even though the velocities of its endpoints are constant. This is indicated by Figure 2, which shows a moving edge (A, B) with different velocities and a moving point $\mathbf{s}$ at sequential times. In this figure, the point is colored red when it intersects the segment, which happens at two different times.

Detecting collisions with a possibly warping segment is handled as follows. Given the time T, vectors $\mathbf{aa}, \mathbf{bb}, \mathbf{ww}, \mathbf{vv} \in \mathbf{R}^2$, and a sign σ (equal to -1 or 1), it is possible to define a concise boolean function `dot_nn_lin` that returns a pair of times in $[0, T]$ (i.e., an interval) such that for every time t in $[0, T]$, $(\mathbf{ww} + t \cdot \mathbf{vv}) \cdot (\mathbf{aa} + t \cdot \mathbf{bb}) \geq 0$ if and only if t is in one of the intervals `dot_nn_lin`$(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, -1)$ or `dot_nn_lin`$(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, 1)$. If no such t exists, then the function returns the empty interval $(T, 0)$. The definition of `dot_nn_lin` is omitted from this paper because it is simple to define and depends mostly on the quadratic formula and considering several special cases (such as the cases where $\mathbf{vv} \cdot \mathbf{bb}$ is greater than, equal to, or less than 0).

The function `dot_nn_lin` is used to define another function `dot_nn` that also has as parameters two signs $\sigma_1 = \pm 1$ and $\sigma_2 = \pm 1$, as well as a positive real number F. The function `dot_nn` also returns an

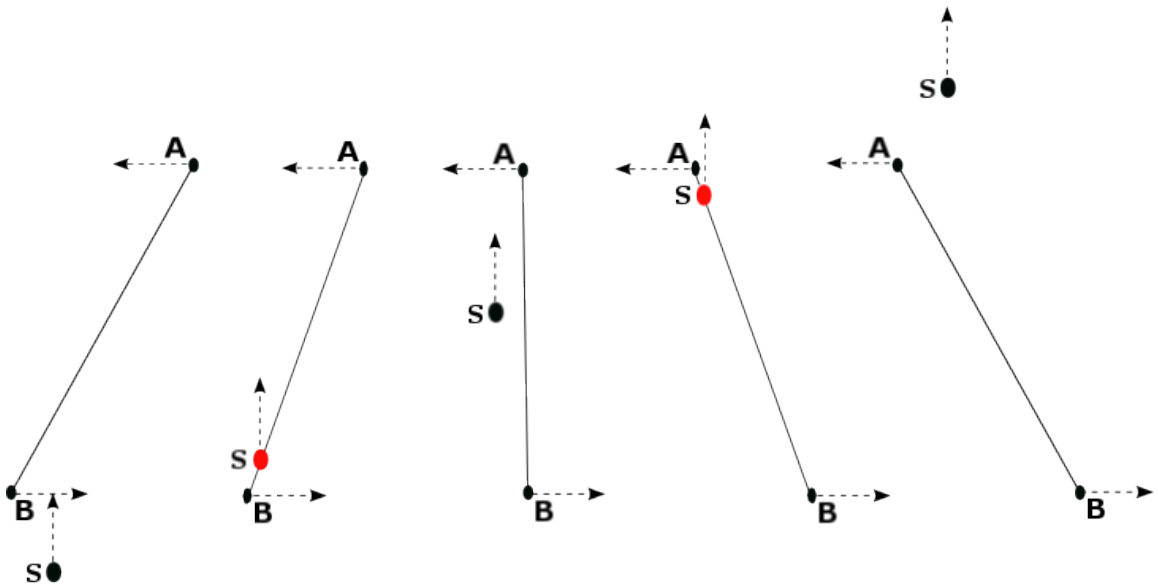

Figure 2. A Linearly Moving Point Intersecting a Warping Edge Twice

interval with endpoints in $[0, T]$, and it may also be empty under certain circumstances. It is defined as follows.

$$\texttt{dot\_nn}(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, \sigma_1, \sigma_2, F) \equiv \texttt{dot\_nn\_lin}(T, F \cdot \mathbf{aa} + \sigma_2 \cdot \mathbf{ww}, F \cdot \mathbf{bb} + \sigma_2 \cdot \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, \sigma_1)$$

The correctness statement, which can be proved mathematically, is stated as follows. Suppose that $(\text{lb}, \text{ub}) = \texttt{dot\_nn}(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, -1, 1, F)$ and $(\text{lbn}, \text{ubn}) = \texttt{dot\_nn}(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, -1, -1, F)$, and also that $(\text{lbx}, \text{ubx}) = \texttt{dot\_nn}(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, 1, 1, F)$ and $(\text{lbnx}, \text{ubnx}) = \texttt{dot\_nn}(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, 1, -1, F)$. For any real number t, it holds that t is in $[0, T]$ and $|(\mathbf{ww} + t \cdot \mathbf{vv}) \cdot (\mathbf{aa} + t \cdot \mathbf{bb})| \leq F \cdot (\mathbf{aa} + t \cdot \mathbf{bb})^2$ if and only if both of the following points hold:

- Either $\text{lb} \leq t \leq \text{ub}$ or $\text{lbx} \leq t \leq \text{ubx}$
- Either $\text{lbn} \leq t \leq \text{ubn}$ or $\text{lbnx} \leq t \leq \text{ubnx}$

Using the function `dot_nn`, it is possible to define a function `edge_detect_simp` that returns true precisely when there exists some time t in the interval $[0, T]$ at which both the following inequalities hold

$$|(\mathbf{ww}+t\cdot\mathbf{vv})\cdot(\mathbf{aa}+t\cdot\mathbf{bb})| \leq (1+F)\cdot\|\mathbf{aa}+t\cdot\mathbf{bb}\|^2 \quad \wedge \quad |(\mathbf{ww}+t\cdot\mathbf{vv})\cdot(\mathbf{aa}^{\perp}+t\cdot\mathbf{bb}^{\perp})| \leq F\cdot\|\mathbf{aa}^{\perp}+t\cdot\mathbf{bb}^{\perp}\|^2 \quad (1)$$

The number F is close to zero (e.g. 0.0001), and its purpose is mitigating floating point computation errors. The function `edge_detect_simp` computes this information using the fact that for any real numbers r and g, $|r| \leq g^2$ if and only if $-r \leq g^2$ and $r \leq g^2$ both hold. Thus, the above inequalities can be expressed in terms of quadratics in t, and the quadratic formula is then used to determine whether, for some $t \leq T$, those inequalities all hold. The following is the process for computing `edge_detect_simp`.

`edge_detect_simp`$(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, F)$**:**

1. For every instantiation of six signs $\epsilon_1, \epsilon_2, \epsilon_3, \sigma_1, \sigma_2, \sigma_3$ (each is -1 or 1), set
   - $(\text{lb}_1, \text{ub}_1) = \texttt{dot\_nn}(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, \epsilon_1, \epsilon_2, 1 + F)$
   - $(\text{lb}_2, \text{ub}_2) = \texttt{dot\_nn}(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}, \mathbf{bb}, \epsilon_3, -\epsilon_2, 1 + F)$

- $(\mathrm{lb}_3, \mathrm{ub}_3) = \texttt{dot\_nn}(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}^\perp, \mathbf{bb}^\perp, \sigma_1, \sigma_2, F)$
- $(\mathrm{lb}_4, \mathrm{ub}_4) = \texttt{dot\_nn}(T, \mathbf{ww}, \mathbf{vv}, \mathbf{aa}^\perp, \mathbf{bb}^\perp, \sigma_3, -\sigma_2, F)$

2. If $\max(\max(\mathrm{lb}_1, \mathrm{lb}_2), \max(\mathrm{lb}_3, \mathrm{lb}_4)) \leq \min(\min(\mathrm{ub}_1, \mathrm{ub}_2), \min(\mathrm{ub}_3, \mathrm{ub}_4))$ for any six signs ϵ_1, ϵ_2, ϵ_3, σ_1, σ_2, σ_3, then return *true*.

3. If no such signs exist, then return *false*

The detection function for warping edges takes as parameters the position $\mathbf{s}$, velocity $\mathbf{v}$, two vertices $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ of the polygon, the two velocities $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$ for these vertices, the lookahead time T, and the number F (a small fraction designed to bound computation errors). It is defined as follows.

$$\texttt{edge\_detect}(T, \mathbf{s}, \mathbf{v}, \boldsymbol{p}_i, \boldsymbol{p}_j, \boldsymbol{v}_i, \boldsymbol{v}_j, F) \equiv \texttt{edge\_detect\_simp}\Big(T,\ s - \Big(\frac{1}{2}\boldsymbol{p}_j + \frac{1}{2}\boldsymbol{p}_i\Big),\ \mathbf{v} - \Big(\frac{1}{2}\boldsymbol{v}_j + \frac{1}{2}\boldsymbol{v}_i\Big),\ \frac{1}{2}\boldsymbol{p}_j - \frac{1}{2}\boldsymbol{p}_i,\ \frac{1}{2}\boldsymbol{v}_j - \frac{1}{2}\boldsymbol{v}_i, F\Big) \tag{2}$$

It has been formally proved that the function `edge_detect` is correct. That is, if the trajectory from $\mathbf{s}$ along $\mathbf{v}$ ever meets one of the edges of the moving (and possibly warping) polygon within time T, then the function will return `true` for that edge. The detection function for polygons with different velocities at each vertex is defined as follows.

`detect_diff_2D`$(T, \mathbf{s}, \mathbf{v}, \boldsymbol{P}, \boldsymbol{V}, F,$ `BUFF`$)$**:**

1. Using the function `definitely_outside`, defined above, check whether the point $\mathbf{s}$ is definitely outside $\boldsymbol{P}$. If not, then return *true*.

2. For each edge from $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, with velocities $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$, calculate `edge_detect`$(T, \mathbf{s}, \mathbf{v}, \boldsymbol{p}_i, \boldsymbol{p}_j, \boldsymbol{v}_i, \boldsymbol{v}_j, F)$. If this returns *true* for any edge, then return *true*.

3. Otherwise, return *false*.

II.J. Well Formed 2D Polygons

It is possible that the polygon $\boldsymbol{P}$ is not well formed for a variety of reasons:

- Two non-adjacent edges may cross.
- A vertex may be very close to a non-adjacent edge.
- A vertex may occur at a very sharp corner.
- The vertices may not be in counterclockwise order.

This section presents a method to determine whether a given polygon $\boldsymbol{P}$ is well formed in the sense that it does not exhibit these problems.

Rather than determining whether two non-adjacent edges may cross, the function that determines whether $\boldsymbol{P}$ is well formed uses the function `segments_close`, defined in Section II.G, to determine if they come within distance `BUFF` of each other. If so, then the polygon is said to be not well formed. Similarly, the function `segments_close`, when used in this manner, ensures that no vertex may be close to a non-adjacent edge.

For determining whether any vertex of $\boldsymbol{P}$ occurs at a very sharp corner, a minimal angle threshold $\kappa > 0$ is chosen. The angle κ can be chosen by the user. Note that if $\mathbf{g}$ and $\mathbf{h}$ are nonzero vectors in $\mathbf{R}^2$, then the angle between $\mathbf{g}$ and $\mathbf{h}$ is less than κ if and only if

$$\mathbf{g} \cdot \mathbf{h} \ < \ -\cos(\kappa) \cdot \|\mathbf{g}\| \cdot \|\mathbf{h}\|.$$

For determining whether the vertices of $\boldsymbol{P}$ are in counterclockwise order, an extremal vertex of $\boldsymbol{P}$ is chosen, and it is determined whether the two adjacent edges of the polygon at that point make a *right*

turn or a *left* turn at that point, a left turn indicating that the vertices are in counterclockwise order. The extremal vertex is the unique vertex $\boldsymbol{p}_i$ such that for every other vertex $\boldsymbol{p}_j$, $\boldsymbol{p}_{jx} \geq \boldsymbol{p}_{ix}$ and $\boldsymbol{p}_{jy} > \boldsymbol{p}_{iy}$. The function that determines if a polygon is well formed is defined below.

`well_formed`$(\boldsymbol{P}, \texttt{BUFF})$**:**

1. For every pair of vertices $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ with $i \neq j$, if $\|\boldsymbol{p}_i - \boldsymbol{p}_j\| < 2 \cdot \texttt{BUFF}$, then return *false*.
2. For every edge from $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, and every other edge from $\boldsymbol{p}_q$ to $\boldsymbol{p}_k$ that is *not adjacent* to the first, if `segments_close`$(\boldsymbol{p}_i, \boldsymbol{p}_j, \boldsymbol{p}_q, \boldsymbol{p}_k, \texttt{BUFF})$ returns *true*, then return *false*.
3. For every edge from $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, and every other edge from $\boldsymbol{p}_q$ to $\boldsymbol{p}_k$ that is *is adjacent* to the first, if $(\boldsymbol{p}_j - \boldsymbol{p}_i) \cdot (\boldsymbol{p}_k - \boldsymbol{p}_q) < -\cos(\kappa) \cdot \|\boldsymbol{p}_j - \boldsymbol{p}_i\| \cdot \|\boldsymbol{p}_k - \boldsymbol{p}_q\|$, then return *false*.
4. Let $\boldsymbol{p}_i$ be the unique vertex such that for every other vertex $\boldsymbol{p}_j$, $\boldsymbol{p}_{jx} \geq \boldsymbol{p}_{ix}$ and $\boldsymbol{p}_{jy} > \boldsymbol{p}_{iy}$. Let $\boldsymbol{p}_{(i-1)^*}$ be the vertex directly *before* $\boldsymbol{p}_i$, and let $\boldsymbol{p}_{(i+1)^*}$ be the vertex directly *after* $\boldsymbol{p}_i$. If $(\boldsymbol{p}_i - \boldsymbol{p}_{(i-1)^*}) \cdot (\boldsymbol{p}_{(i+1)^*} - \boldsymbol{p}_i)^{\perp} \leq 0$, then return *false* because the polygon turns *right* at that point.
5. Otherwise, return *true*.

III. Detection for 3D Polygons Using `detect_3D`

For detecting collisions between a point and a 3D polygon, these polygons are modeled as 2D polygons with a minimum height and a maximum height (which can be used to model altitude). As in Section II, each 2D polygon is specified simply as a sequence $\boldsymbol{P} = \{\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n\}$ of points in the horizontal plane $\mathbf{R}^2$, each having an x and a y coordinate. If it is moving, then there is a sequence $\boldsymbol{V} = \{\boldsymbol{v}_0, \ldots, \boldsymbol{v}_n\}$ of velocities in $\mathbf{R}^2$, one velocity for each vertex of $\boldsymbol{P}$. There is also a vertical speed for each 3D polygon, which may be simply 0. The moving point itself is modeled as a 2D point $\mathbf{s}$ in Euclidean space $\mathbf{R}^2$, along with a height (altitude) $\mathtt{s}_z$ at time 0. The inputs to the detection function between a point and a 3D polygon are therefore as follows.

- An initial horizontal position $\mathbf{s}$ in 2D Euclidean space $\mathbf{R}^2$
- An initial horizontal velocity $\mathbf{v}$ in $\mathbf{R}^2$
- A initial height $\mathtt{s}_z$ for the point
- A vertical speed $\mathtt{v}_z$ for the point
- A sequence $\boldsymbol{P} = \{\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n\}$ of points in $\mathbf{R}^2$
- A sequence $\boldsymbol{V} = \{\boldsymbol{v}_0, \ldots, \boldsymbol{v}_n\}$ of velocities in $\mathbf{R}^2$
- A minimum height `pmin` and maximum height `pmax` for the polygon
- A vertical speed `vsp` for the polygon
- A positive lookahead time T
- A buffer distance `BUFF` and possibly a small fraction F (see Section II.I)

The detection method for 3D polygons returns *true* if, during the time interval $[0, T]$, there exists any time $t \in [0, T]$ such that the following both hold.

- The horizontal point $\mathbf{s} + t \cdot \mathbf{v}$ is inside the 2D polygon $\boldsymbol{P} + t \cdot \boldsymbol{V}$.
- The height (e.g., altitude) $\mathtt{s}_z + t \cdot \mathtt{v}_z$ is inside the interval $[\texttt{pmin} + t \cdot \texttt{vsp}, \texttt{pmax} + t \cdot \texttt{vsp}]$.

The detection method therefore works as follows.

detect_3D($T, \mathbf{s}, \mathbf{v}, \mathtt{s}_z, \mathtt{v}_z, \boldsymbol{P}, \boldsymbol{V}, \mathtt{pmin}, \mathtt{pmax}, \mathtt{vsp}, \mathtt{BUFF}, F$):

1. Calculate the exact subinterval $[a, b]$ of $[0, T]$ where $sz + t \cdot \mathtt{v}_z$ is inside the interval $[\mathtt{pmin} + t \cdot \mathtt{vsp}, \mathtt{pmax} + t \cdot \mathtt{vsp}]$ if and only if t is in $[a, b]$. If no such time exists, then $[a, b]$ can be set to $[T, 0]$.

2. Use a 2D polygon collision detection function, such as `detect_same_2D` or `detect_diff_2D` (defined in Sections II.H and II.I), with initial horizontal position $\mathbf{s} + a \cdot \mathbf{v}$, initial 2D polygon $\boldsymbol{P} + a \cdot \boldsymbol{V}$, and lookahead time $b - a$ to determine if there is any time t in $[a, b]$ where $\mathbf{s} + t \cdot \mathbf{v}$ is inside the 2D polygon $\boldsymbol{P} + t \cdot \boldsymbol{V}$. If so, the method returns *true* and otherwise *false*.

Computing the time interval $[a, b]$ in step (1) of this method is straightforward and involves only a few cases and basic mathematics. Therefore, all that is needed to accomplish 3D collision detection for polygons is a 2D collision detection algorithm.

IV. Conclusion

Weather avoidance systems are safety critical due to the potential dangers of convective weather (see FAA Advisory Circular 00-24C[1]). The significant disruptions caused by bad weather present a challenge due to the uncertain behavior of weather systems.[2] Any viable air traffic management system must reliably integrate weather awareness and avoidance.

This paper addresses this issue by describing functions for determining whether a linearly moving point, such as an aircraft, will intersect a polygonal region, such as a cell of convective weather, within a predetermined lookahead time. The algorithms work for a 3 dimensional polygon formed from a 2 dimensional polygon and maximum/minimum altitudes. The algorithms are being developed for inclusion in NASA's Stratway software tool, which allows low-fidelity air traffic management concepts to be easily prototyped and quickly tested. In addition, several, but not all, of the detection functions defined in this paper have been mathematically proved to be correct. Most importantly the functions `segments_close` and `edge_detect`, which are the main functions called in the detection algorithms `detect_same_2D` and `detect_diff_2D`, have been proved correct, assuming there are no floating point errors. Such proofs are useful for safety critical software systems because they provide improved assurance of correctness over standard testing methods.

References

[1]The Federal Aviation Administration. Advisory circular 00-24c. (00-24C), 2 2013.

[2]M.C. Consiglio, J.P. Chamberlain, and S.R. Wilson. Integration of weather avoidance and traffic separation. In *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pages 3B4–1–3B4–14, Oct 2011.

[3]Thomas C. Hales. The Jordan Curve Theorem, Formally and Informally. *The American Mathematical Monthly*, 114(10):882–894, 2007.

[4]M. C. Lin, D. Manocha, J. Cohen, and S. Gottschalk. Collision Detection: Algorithms and Applications. pages 129–142. A K Peters, 1997.

[5]Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceeding of the 11th International Conference on Automated Deductioncade*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, June 1992.

[6]Mikhail Rubnich and Rich DeLaura. *An Algorithm to Identify Robust Convective Weather Avoidance Polygons in En Route Airspace*. American Institute of Aeronautics and Astronautics, 2015/08/20 2010.

[7]Stefan Schirra. How reliable are practical point-in-polygon strategies? In Dan Halperin and Kurt Mehlhorn, editors, *Algorithms - ESA 2008*, volume 5193 of *Lecture Notes in Computer Science*, pages 744–755. Springer Berlin Heidelberg, 2008.

[8]G. van den Bergen. *Collision Detection in Interactive 3D Environments*. Interactive 3D technology series. Taylor & Francis, 2004.